COMSC-210 Lecture Topic 7
Stacks and Queues

□ Reference

Childs Ch. 8

□ Stacks (LIFO)

Last In, First Out
developed in the late 1950's
  for expression evaluation
specification:
  push (add a value)
  pop (remove newest value)
  peek (look at newest value)
  top (same as peek)
  size (#items in the stack)
  empty (is it empty?)
  clear (remove all values)
applications
  recursive solutions
  RPN calculations
  games with playing cards
  subprogram call tracking
  HTML tag processing
the C++ STL `stack`

```
bool empty() const
int size() const
T& top() // mutable reference
void pop()
void push(const T&)
```

□ Queues (FIFO)

First In, First Out
represents waiting in line
specification:
  push (add a value)
  pop (remove *oldest* value)
  peek (look at *oldest* value)
  top (same as peek)
  size (#items in the queue)
  empty (is it empty?)
  clear (remove all values)
applications
  server simulations
  tracking user requests
  recursive solutions
the C++ STL `queue`

```
bool empty() const
int size() const
T& front() // oldest value, mutable reference
```

□ Array Implementation Of Stack (continued)

```
Stack<T>::Stack(int init_cap) // function
{
  cap = init_cap;
  values = new T[cap]);
  siz = 0; // initally empty
}

bool empty() const {return 0 == siz;} // inline
int size() const {return siz;} // inline
void clear() {siz = 0;} // inline

void Stack<T>::push(const T& parameter)
{
  if (++siz > capacity)
    double the capacity
  values[siz - 1] = parameter;
}

bool Stack<T>::peek(T& parameter) const
{
  if (0 == siz) return false; // failed
  parameter = values[siz - 1];
  return true; // success
}

bool Stack<T>::pop(T& parameter)
{
  if (0 == siz) return false; // failed
  parameter = values[--siz];
  if (siz > 2 && siz < capacity / 4)
    halve the capacity
  return true; // success
}
```

□ Linked-List Implementation Of Stack

private class member

```
Node* start; // no end needed
int siz; // track size
```

```
Stack<T>::Stack()
{
  start = 0; // empty list
  siz = 0;
}

bool empty() const {return 0 == siz;} // inline
int size() const {return siz;} // inline

void Stack<T>::push(const T& parameter)
{
  Node* node = new Node;
  node->value = parameter;
  node->next = start;
  T& end() // last-added value, mutable reference
  void pop() // lose oldest value
  void push(const T& )
```

□ Inspecting Stacks And Queues

- no interface provided for seeing
  - anything other than end values
  - that is, no `operator[]`
- common solution -- the "copy-pop" method
  - copy the stack (or queue)
  - peek/pop the copy until empty
  - discard the copy

```
stack<int> s;
...
for (stack<int> cpy = s; !cpy.empty(); cpy.pop())
  cout << cpy.top();
```

□ Implementation As Array Or Linked-List

- stacks are easy either way
  - as list, insert/delete at start
  - as array, insert/delete at end
    - track index of end
    - expand/shrink as necessary
- queues easy as lists
  - insert at *end*
  - "need" pointer to track end
- queues *not* easy as arrays
  - need to track start *and* end
  - need to handle wrap-around

□ Design Considerations

- error codes vs `bool` returns
- linked list node `struct`

```
struct Node
{
  T value;
  Node* next;
};
```

no `inUse` because ALL are in use

- array-based option
  - use DynamicArray template or not

- linked queue option
  - use header node or not
- dynamic memory management requirements
  - destructor
  - copy constructor
  - overloaded operator=

- pop options:
  - return nothing (void)

```
  start = node;
  ++siz;
}

bool Stack<T>::peek(T& parameter) const
{
  if (0 == siz) return false; // failed
  parameter = start->value;
  return true; // success
}

bool Stack<T>::pop(T& parameter)
{
  if (0 == siz) return false; // failed
  parameter = start->value;
  Node* p = start->next;
  delete start;
  start = p;
  --siz;
  return true; // success
}

void Stack<T>::clear()
{
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
  }
  siz = 0;
}
```

□ Linked-List Implementation Of Queue

private class members

```
  Node* start;
  Node* end; // for efficiency
  int siz; // track size
```

```
Queue<T>::Queue()
{
  start = 0;
  end = 0;
  siz = 0;
}

bool Queue<T>::empty() const {return 0 == start;}

void Queue<T>::push(const T& parameter)
{
  Node* node = new Node;
  node->value = parameter;
  node->next = 0;
  if (end) end->next = node;
  else start = node;
  end = node;
  ++siz;
}

bool Queue<T>::peek(T& parameter) const
{
  if (0 == start) return false; // failed
```

- return copy
- copy to non-const parameter

push options for new value:

- parameter passed by const reference
- parameter passed by value

peek options:

- return copy
- return reference
- return const reference
- copy to non-const parameter

□ Array Implementation Of Stack

private class members

```
  static const int INIT_CAP = 100;
  int cap; // capacity
  T* values;
  int siz; // track size

// prototype with default initial capacity
Stack(int=INIT_CAP);
```

```
  parameter = start->value;
  return true; // success
}

bool Queue<T>::pop(T& parameter)
{
  if (0 == start) return false; // failed
  parameter = start->value;
  Node* p = start->next;
  delete start;
  start = p;
  if (start == 0) end = 0;
  --siz;
  return true; // success
}

void Queue<T>::clear()
{
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
  }
  end = 0;
  siz = 0;
}
```

Dynamic Memory Management For Array-Based And Linked-List-Based Data Structures

Destructor For Arrayed Stack

```
~Stack(){delete [] values;} // inline
```

Destructor For Linked Stack

```
template <class T>
Stack<T>::~Stack()
{
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
} }
```

Destructor For Linked Queue

```
template <class T>
Queue<T>::~Queue()
{
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
} }
```

Copy Constructor For Arrayed Stack

```
template <class T>
Stack<T>::Stack(const Stack<T>& a)
{
  cap = a.cap;
  siz = a.siz;
  values = new T[cap];
  for (int i = 0; i < cap; i++)
    values[i] = a.values[i];
}
```

Copy Constructor For Linked Stack

```
template <class T>
Stack<T>::Stack(const Stack<T>& a)
{
  start = 0;
  Node* end = 0; // temporary end pointer
  siz = a.siz;
  for (Node* p = a.start; p; p = p->next)
```

Copy Constructor For Linked Queue

```
template <class T>
Queue<T>::Queue(const Queue<T>& a)
{
  start = 0;
  end = 0;
  siz = a.siz;
  for (Node* p = a.start; p; p = p->next)
  {
    Node* node = new Node;
    node->value = p->value;
    node->next = 0;
    if (end) end->next = node;
    else start = node;
  {
    Node* node = new Node;
    node->value = p->value;
    node->next = 0;
    if (end) end->next = node;
    else start = node;
    end = node;
} }
```

```
    end = node;
} }
```

Operator= For Arrayed Stack

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& a)
{
  if (this == &a) return *this;

  delete [] values;
  cap = a.cap;
  siz = a.siz;
  values = new T[cap];
  for (int i = 0; i < cap; i++)
    values[i] = a.values[i];

  return *this;
}
```

Operator= For Linked Stack

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& a)
{
  if (this == &a) return *this;

  // deallocate existing queue
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
  }

  // build new queue
  Node* end = 0; // temporary end pointer
  for (Node* p = a.start; p; p = p->next)
  {
    Node* node = new Node;
    node->value = p->value;
    node->next = 0;
    if (end) end->next = node;
    else start = node;
    end = node;
  }
  siz = a.siz;

  return *this;
}
```

Operator= For Linked Queue

```
template <class T>
Queue<T>& Queue<T>::operator=(const Queue<T>& a)
{
  if (this == &a) return *this;

  // deallocate existing queue
  while (start)
  {
    Node* p = start->next;
    delete start;
    start = p;
  }

  // build new queue
  end = 0; // data member end pointer
  for (Node* p = a.start; p; p = p->next)
  {
    Node* node = new Node;
    node->value = p->value;
    node->next = 0;
    if (end) end->next = node;
    else start = node;
    end = node;
  }
  siz = a.siz;

  return *this;
}
```

Resizing For Arrayed Stack (private function)

```
template <class T>
void Stack<T>::changeCapacity(int newCap)
{
  T* temp = new T[newCap];
  int limit = newCap > cap ? cap : newCap;
  for (int i = 0; i < limit; i++) temp[i] = values[i];
  delete [] values;
  values = temp;
  cap = newCap;
}
```